Appearing in: Proceedings of SPIE Vol. #5097 Geo-Spatial and Temporal Image and Data Exploitation III, AeroSense Technologies and Systems for Defense and Security, 21-25 April 2003, Orlando, Florida USA.

Precision Markup Modeling and Display in a Global Geospatial Environment

Zachary Wartell[a], William Ribarsky[a], and Nickolas Faust[b]

[a]Graphics, Visualization, and Usability Center, Georgia Institute of Technology
[b]Center for GIS and Spatial Analysis Technologies, Georgia Institute of Technology

ABSTRACT

A data organization, scalable structure, and multiresolution visualization approach is described for precision markup modeling in a global geospatial environment. The global environment supports interactive visual navigation from global overviews to details on the ground at the resolution of inches or less. This is a difference in scale of 10 orders of magnitude or more. To efficiently handle details over this range of scales while providing accurate placement of objects, a set of nested coordinate systems is used, which always refers, through a series of transformations, to the fundamental world coordinate system (with its origin at the center of the earth). This coordinate structure supports multi-resolution models of imagery, terrain, vector data, buildings, moving objects, and other geospatial data. Thus objects that are static or moving on the terrain can be displayed without inaccurate positioning or jumping due to coordinate round-off. Examples of high resolution images, 3D objects, and terrain-following annotations are shown.

Keywords: high resolution imagery, 3D detail, modeling, precision markup, interactive visualization

1. INTRODUCTION

This paper describes VGIS, a global geospatial 3D visualization tool, and its ability to provide precision markup and display of geospatial information over geometric scales covering 10 orders of magnitude. VGIS stands for Virtual Geographic Information System. When we began this project several years ago [5][8], our premise was that GIS and 3D visualization are intimately related and that by merging them one can empower and enlarge the other. The merger enlarges and empowers GIS because it offers a fully 3D GIS that can be interactively explored and displayed. The merger enlarges and empowers interactive visualization because it gives visualization concrete and meaningful applications. We continue to expand the system toward scalable, multi-resolution data organizations for stationary terrestrial objects[2], volumetric Doppler radar weather[4], semi-automated building extraction [Gri01][13], and polyline vector data[12]. We developed and continue to develop interfaces for alternative display environments such as virtual reality[11] and wearable computing [6][7]. A key goal of VGIS is to maintain the ability to show all types of data together in a single software system where the user can gain new insights into correlations between these various data types. This requires accurate registration when converting the raw data into efficiently rendered primary and secondary memory data structures and requires accurate rendering of the varying data types. To provide this rendering accuracy for global geospatial data and provide accurate positioning down to millimeters, VGIS uses a set of nested coordinate systems which always refer, through a series of transformations, to a the fundamental Earth-Center-Earth-Fixed world coordinate system. This paper describes the components of VGIS architecture that provide accurate rendering of geospatial data and related geometric objects.

2. VGIS ARCHITECTURE

This section presents an overview of VGIS's architecture, focusing on its management of rendered geometry precision. VGIS uses a multi-threaded threaded architecture (Figure 1). The threads of key importance to this paper are the render thread and the manager threads: terrain manager, object manager, and volume manager. In general the manager threads perform LOD computations and possibly request the dynamic loading of data as it comes into view. The manager threads issue VGIS display list calls whose function names are prefixed by *dl_*. Many of these calls are named directly

after OpenGL calls such as *dl_glVertex3fv(DisplayList* dl, GLfloat v[3]).* The *dl_* calls marshal their arguments and place them onto a triple buffered display list that is read and processed asynchronously by the Render Thread. The Render Thread is the only thread that issues OpenGL calls. On single processor machines the Render Thread main loop generally runs at faster iterations rates (i.e. the frame rate) than the iteration rates of the manager threads. On multi-processor machines, these manager threads can run on separate CPU's. Generally the manager threads do not perform frustum culling; rather they issues special *dl_* frustum culling calls that specify axis-aligned bounding boxes that bound portions of the display list. The render thread then performs the culling tests.

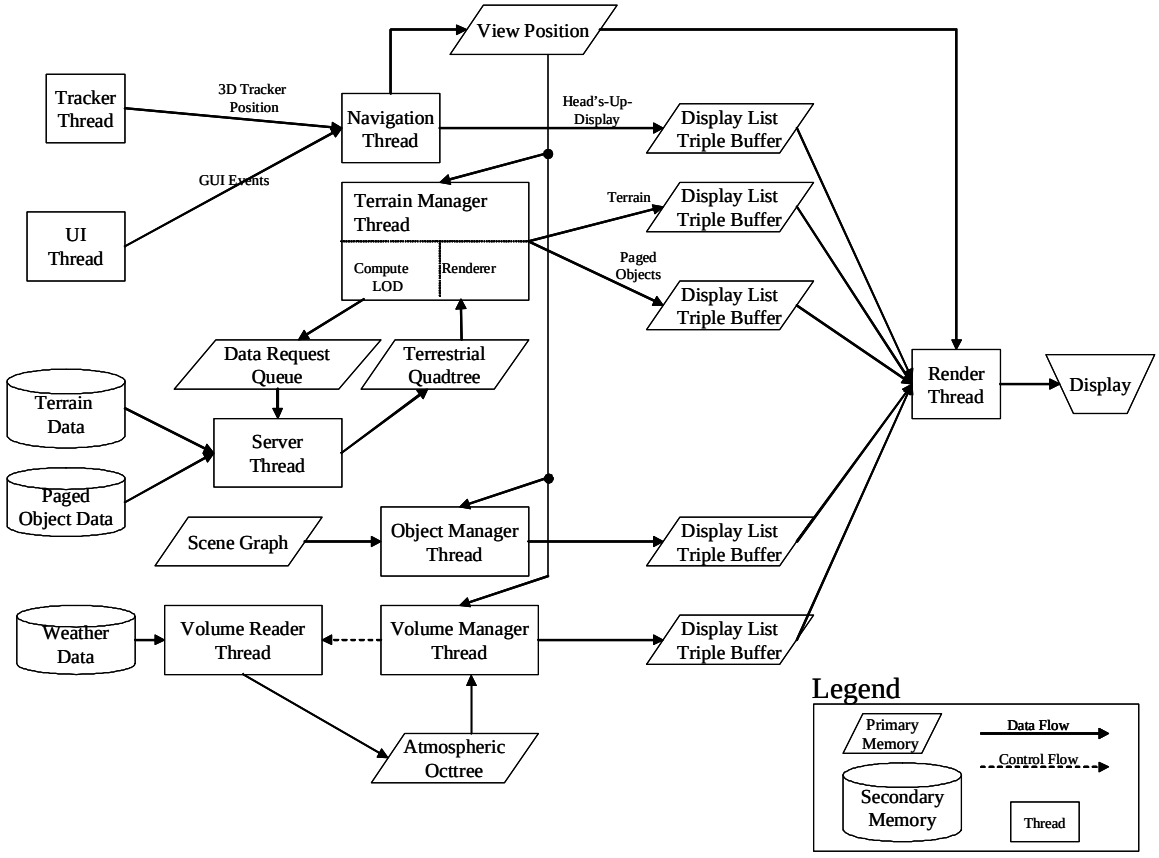


Figure 1: High Level Design of VGIS

The terrain manager renders (to a display list) terrain[8], polyline vector data[12] and paged objects[2]. The terrain consists of geometry and imagery which exist in the secondary storage at independent and varying resolutions. Polyline vector data represent roads, political borders, etc. Paged objects are stationary objects such as buildings and trees. In primary memory, terrain and paged objects are organized into a pointer based terrestrial quadtree[10]. Next, the object manager renders objects from a separate scene graph. Scene graph objects are not inserted into the terrestrial quadtree structure. The volume manager renders volumetric Doppler radar weather data. In primary memory this data is organized into a second spatial tree, the atmospheric octree. (Note, the atmospheric octree tree is not a strict octree. The nodes in the tree may be of degree 2, 4, or 8.)

For the terrestrial quadtree rather than having a single quadtree represent the entire globe, we subdivide the globe into a small number of geodetic zones. In the current implementation, thirty-two $45^o \times 45^o$ zones are used. The number and extent of zones were chosen based on empirical observations of memory requirements, paging overhead, geometric accuracy, etc. Each zone is further subdivided and organized by a hierarchical quadtree structure. A node in a quadtree

corresponds to a raster tile of fixed dimensions and lat/lon resolution according to the level on which it appears in the quadtree. Quadnodes are identified by "quadcodes," which are built in a manner similar to the indices of representations of binary trees, that is, the children of a node with quadcode q are identified by $4q + 1$ through $4q + 4$. In addition, the quadcode contains a quadtree identifier which allows each quadcode to uniquely identify an area on the globe.

In addition to spatially organizing the data, the quadtrees also define the boundaries of local coordinate systems. If a single, geocentric coordinate system were used, and assuming 32-bit single precision floating point is used to describe object geometries, the highest attainable accuracy on the surface of the Earth is half a meter. Clearly, this is not sufficient to distinguish features with details as small as a few centimeters, e.g. the treads on a tank. Visually, this lack in precision results in "wobbling" as the vertices of the geometry are snapped to discrete positions during the graphics pipeline geometric computations[1]. This problem is present in other large scale terrain systems such as T_Vision [Gru95]. To overcome this problem; we define a number of local coordinate systems over the globe which have their origins displaced to the (oblate) spheroid surface that defines the Earth sea-level[5][9]. The origins of the top-level coordinate systems are placed at the geographic centers (i.e. the mean of the boundary longitudes and latitudes) of the quadtree roots. While the centroid of the terrain surface within a given zone would result in a better choice of origin in terms of average precision, we decided for simplicity to opt for the geographic center, noting that the two are very close in most cases. The *z* axis of each coordinate system is defined as the outward normal of the surface at the origin, while the *y* axis is parallel to the intersection of the tangent plane at the origin and the plane described by the North and South poles and the origin. That is, the *y* axis is orthogonal to the *z* axis and locally points due North. The *x* axis is simply the cross product of the *y* and *z* axes, and the three axes form an orthonormal basis. This choice of orientation is very natural as it allows us to approximate the "up" vector by the local *z* axis, which further lets us treat the height field as a flat-projected surface with little error. Hence, the height field LOD algorithm, which is based on vertical error in the triangulation, does not have to be modified significantly to take the curvature of the Earth into account. However, the delta values (see Lindstrom et al.[8]) must be computed in Cartesian rather than geodetic coordinates to avoid over-simplification of constant-elevation but curved areas such as oceans. Figure 2B shows the local coordinate systems for a few zones.

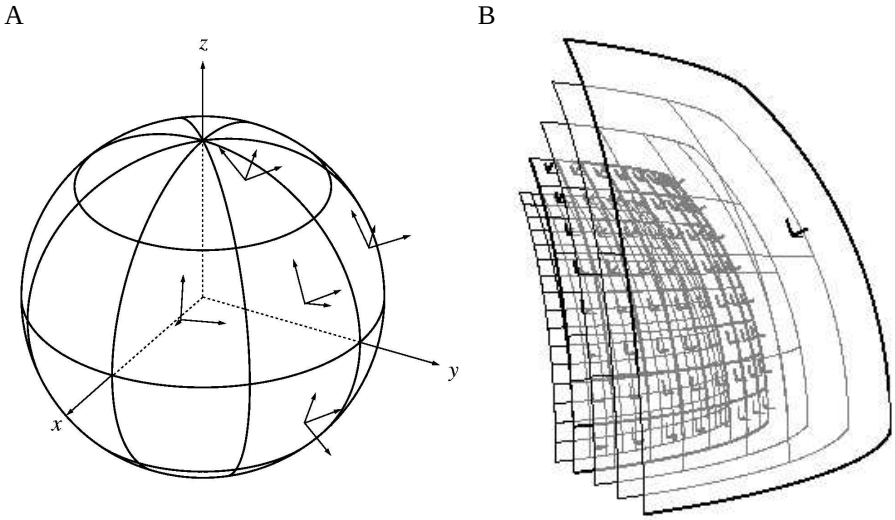


Figure 2 (A) Local coordinate systems for the quadtree roots. The labeled axes correspond to the conventional Earth Centered, Earth Fixed Cartesian XYZ global coordinate system. (B) Illustration of nested coordinate systems in a quadtree. 8 x 8 smaller coordinate systems appear 3 levels below the root node. (VGIS actually has 256 x 256 coordinate systems 8 levels below the root node).

Using the above scheme, the resulting worst case precision for a 45^o x 45^o zone is 25 cm---not significantly better than the geocentric case. We could optionally use a finer subdivision with a larger number of zones to obtain the required precision. However, this would result in a larger number of quadtrees, which is undesirable since the lowest resolution data that can be displayed is defined by the areal extent of the quadtree roots. Hence, too much data would be needed to display the lowest resolution version of the globe. Instead, we define additional coordinate systems within each quadtree.

In the current implementation, we have added 256 x 256 coordinate systems within each quadtree---one coordinate system per node, eight levels below each root node---resulting in a 1 mm worst case precision. Each CCC cell can cover roughly 19.5x19.5x19.5 km of space about the coordinate system origin while maintaining this accuracy. Figure 2B illustrates the idea but with a smaller number of nested coordinate systems.

VGIS uses the following coordinate system naming conventions. The 64-bit ECEF (Earth-Centered Earth Fixed) Cartesian coordinate system is called XYZ64 coordinates. XYZ64 coordinates are in meters. Global geodetic 64-bit coordinates using WGS84 datum as the default are called LLH64 coordinates for Latitude-Longitude-Height. LLH64 coordinates are in degrees and meters. The small Cartesian coordinate systems placed on the spheroid surface are called CCC for Cell Cartesian Coordinates. CCC coordinates are in meters. Both 32-bit and 64-bit CCC coordinate structures are available although typically only CCC32 is used. A CCC coordinate system itself is typically specified by a 64-bit matrix that specifies the mapping from that system to XYZ coordinates (CCC-to-XYZ). We are in the process of implementing a second set of cell based Cartesian Coordinates for the atmospheric octree.

This coordinate system organization differs from other approaches to the precision problem found in other interactive 3D graphics toolkits. Spline[1] always keeps individual objects associated with a "locale." A locale is a 32-bit coordinate system and it is assumed that before objects stray too far from a locale's origin the object will move into a different locale at which point the system transforms the objects location information to be relative to the new locale. The location coordinate is henceforth maintained the relative to the new locale. Spline does not contain a global coordinate system. In contrast, VGIS's CCC coordinate systems are specified relative to the global XYZ coordinate system using 64-bit precision. Java 3D 1.2 [14] has a mechanism similar to Spline's locales. Java 3D has three related classes: VirtualUniverse, Locale, and HiResCoord. The VirtualUniverse is a top-level container for all scene graphs. It consists of a set of Locale objects each of which has its position specified by a HiResCoord object. HiResCoord specifies a 256 fix point location (with the decimal point at bit 128). This "is sufficient to describe a universe in excess of several billion light years across, yet still define objects smaller than a proton." A Java3D Locale only has a location. It has no orientation or scale is present. As a general approach this is quite nice. However, performing operations on 256 bit fix point numbers are costly relative to the 64-bit floating point numbers that VGIS uses to position CCC coordinate system relative to the XYZ system. Also note orientation is important to the VGIS terrain LOD algorithm so that it can orient the CCC coordinate system such the z vector is a good approximation to the height field direction. Java 3D's HiResCoord mechanism wouldn't allow this. Further, the 64-bit host CPU computation of CCC-to-view = CCC-to-world * world-to-view done prior to rendering a CCC system's associated geometry (see below) is more efficient than trying to do similar operations on a 256 fixed-point matrix. Since using 64-bit floats is sufficient for a global terrain system it is the most appropriate solution to the precision problem in VGIS.

Next, we discuss the procedural aspects of using CCC coordinate systems. When the terrain manager renders the terrain vertices and polyline vector data for a given quadnode, it starts with a *dl_loadViewAndMultMatrix* display list command using a matrix argument equal to the quadnode's 64-bit CCC-to-XYZ transform. When the render thread processes this command, the render thread computes the matrix, CCC-to-view = CCC-to-world * world-to-view, using a 64-bit matrix operation on the host CPU. This CCC-to-view transform is then converted to a 32-bit matrix which is pushed onto the OpenGL stack. In primary memory, VGIS stores the coordinates of terrain vertices and vector data polylines for a given quadnode in the quadnode's 32-bit CCC coordinates. These 32-bit coordinates are passed through the render thread to OpenGL. Since the CCC-to-view translation component will be fairly small for quadnodes close to the view point, vertex wobbling does not appear.

For the paged objects associated with a given quadnode, the terrain manager also issues a *dl_loadViewAndMultMatrix* command with the quadnode's CCC-to-XYZ transform. It then proceeds to render (to the display list) the quadnode's paged objects. Unlike the terrain vertices, however, the paged object's vertex coordinates are stored in each object's local coordinate system in 32-bit precision. An object's local coordinate system is stored as a 64-bit matrix mapping local coordinates to the XYZ system. When the terrain manager first renders a paged object, it computes the transform from its local coordinates to the CCC coordinates of the quadnode. This is done using 64-bit matrix operations on the host CPU. (Note, the 64-bit local-to-CCC transform is generally only computed at load time since paged objects do not move.) This local-to-CCC transform is then converted to a 32-bit matrix and cached for the particular object. The 32-bit local-to-CCC matrix is then passed to the render thread. The render thread will then push local-to-CCC onto the

OpenGL stack where OpenGL composes it with the CCC-to-view matrix using OpenGL's internal floating point operations. For quadnodes near the view point, both these matrices (local-to-CCC and CCC-to-view) have relatively small translations which avoids vertex wobbling. The objects 32-bit vertices are passed to OpenGL in their local coordinate system coordinates.

Objects in the object scene graph are rendered by the object manager thread and not by the terrain manager. Scene graph objects are allowed to move. The object API accesses object structures called *obj3d_t*. The position is accessed either as LLH or XYZ coordinates. Internally, the *obj3d_t*'s local-to-world transform is stored as a 64-bit matrix. Each *obj3d_t* has an internal hierarchy of *obj3dnode_t* which represent the articulated parts of an *obj3d_t*. The locations of *obj3d_t* in the XYZ coordinates are expected to roam over the globe. When the object manager renders an *obj3d_t* it calls *dl_loadViewAndMultMatrix* with the objects 64-bit local-to-XYZ position matrix. Again, this will cause the render thread to compose local-to-eye = local-to-XYZ * XYZ-to-eye using 64-bit host CPU operations; convert local-to-eye to 32-bits and call glLoadMatrixf to load this onto the stack. When rendering the articulated sub-components (*obj3d_node_t*'s), the object manager passes regular 32-bit matrices to OpenGL through the display list. This is acceptable because the transforms in the part hierarchy for a single object are assumed to contain relatively small translations. The object vertices are passed in the *obj3dnode_t* local coordinates. The object scene graph has been used for rendering vehicles, 3D icons, and extracted discrete weather features such as mesocyclones.

We have also developed a travel interface for VGIS on the virtual workbench[11]. VR systems require a viewing model capable of accurately representing the position and orientation of the display surfaces, the user's eye points, and 6DOF interaction devices. This is best modeled by a coordinate system hierarchy such as Figure 3. The Platform coordinate system is manipulated to fly the virtual viewpoint through the world. The Tracker is the physical coordinate system of the tracking hardware. The head-sensor is the position of the tracking sensor worn on the users head. The Projection Plane coordinate system embeds the geometric representation of the display screen. VR applications typically show various heads-up-display information such as 2D and 3D compasses. Additionally, the hand held 6DOF devices typically have some virtual 3D object representation it such as a virtual laser pointer or virtual hand. The view coordinate system should exist in the scene graph like any other geometric object. Logically the just mentioned graphical objects hang off these view coordinate systems in the scene graph. A naive implementation, however, can lead to severe vertex wobbling of these view hierarchy attached objects. This occurs when the Platform translation component grows large in XYZ space. The problem is exacerbated since in a VR systems like the virtual workbench, scale must be manipulated as a separate 7^{th} degree of freedom to effectively use head-tracked stereo display and 6DOF devices. To avoid rendering precision problems for the view hierarchy attached objects, the object manager looks for the view Platform object when traversing the scene graph. When the object manager encounters the Platform object, it makes a special call, *dl_loadPlatformViewMatrix*, before proceeding to traverse render the rest of the objects attached to the view hierarchy. When the render thread processes this command it computes *platform-to-eye* directly using explicit knowledge of the view hierarchy without explicitly composing *platform-to-xyz* * *xyz-to-eye*. The render thread then loads *platform-to-eye* onto the OpenGL matrix stack. This completely avoids having deal with the large translation component found in *platform-to-XYZ*. The result is that no vertex wobbling occurs for objects carried by the view hierarchy regardless of view position and scale in the virtual world.

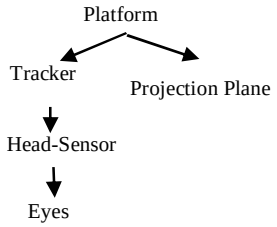
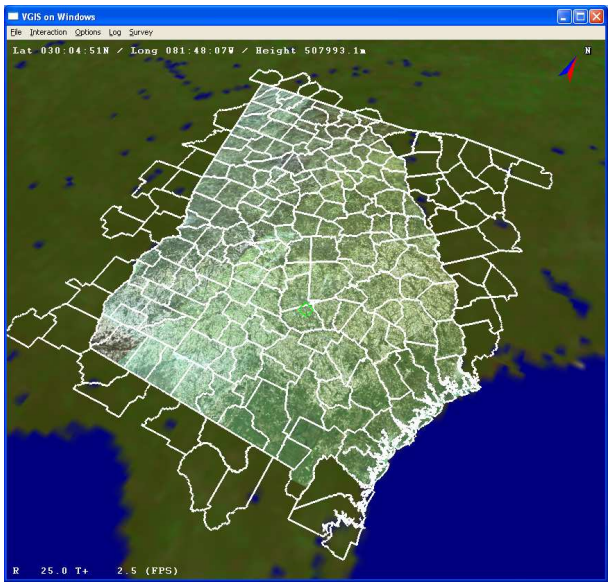


Figure 3: Example coordinate system hierarchy for head-tracked VR display with a stationary display surface

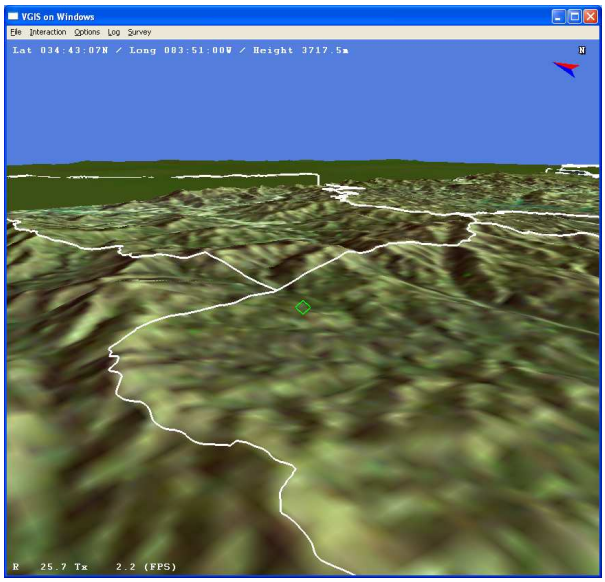
3. PRECISION MARKUP EXAMPLES

The figures on the following page illustrate various examples of objects and terrain rendering using these coordinate systems. Figure 4A through C illustrate polyline vector data of the counties of the state of Georgia. B and C show a textured and wireframe view. The original polyline data consisted of 240K points. The terrain database consists of 50 km elevation data for the world at large; 30 m elevation data for Georgia and 10 m data for downtown Atlanta. Figure D illustrates the county borders rendered as terrain following fences instead of line segments. Figure 5A and B illustrates Georgia Tech campus. Figure B contains a few 3D icons on the terrain. The buildings are paged objects and the icons are objects in the object scene graph. Figure 5C illustrates the GPS point locations traveled by a student as he carries a laptop running GPS-enabled VGIS. Figure 5D shows a colored coded mesocyclone, a wind shearing phenomenon, and its velocity vector. The mesocyclone was extracted from the raw Doppler radar information using a weather analysis tool (WDSS II) developed by collaborators at the National Severe Storm Laboratory. Figure 5E shows some interactive markup of the terrain.

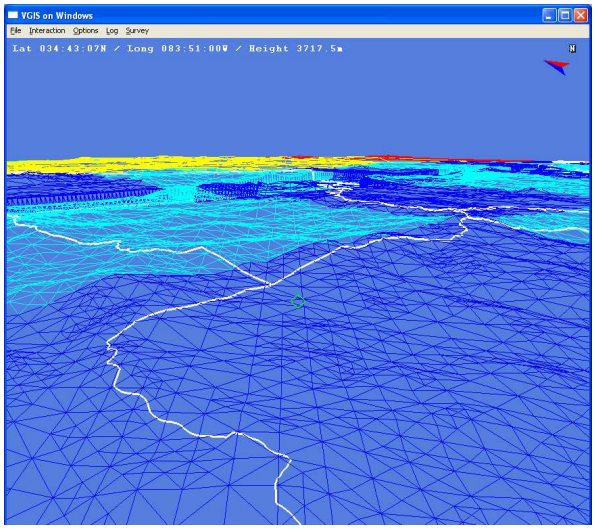
A



B



C



D

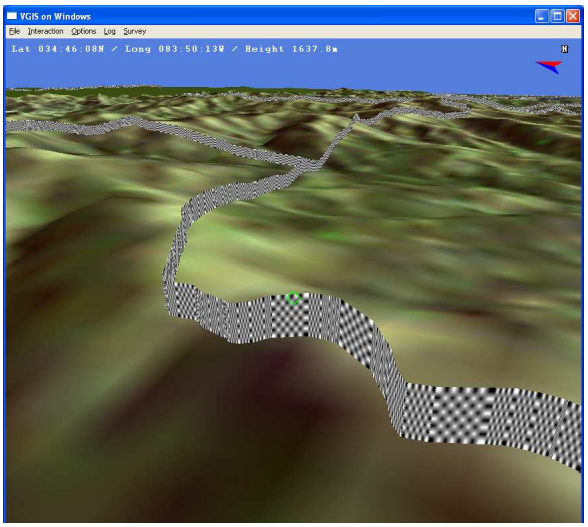


Figure 4: (A)-(C) Polyline Vector Data rendered as terrain following lines, (D) polyline vector data rendered as raised “fences”

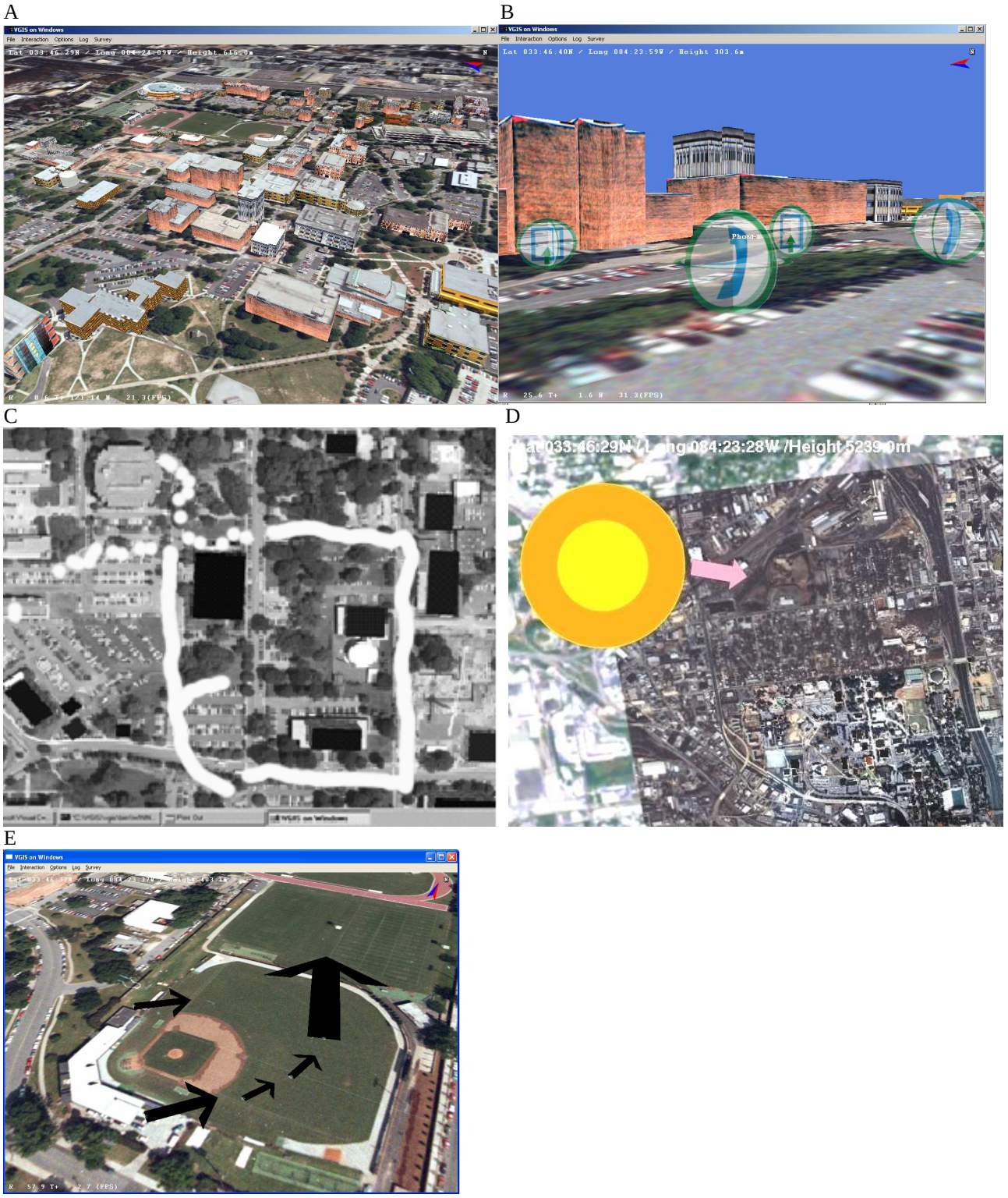


Figure 5: (A) Georgia Tech Campus (B) 3D Icons on terrain (C) real-time GPS trace of a user wandering Georgia Tech campus with GPS equipped VGIS on laptop mobile (D) Mesocyclone and velocity vector (E) Interactive markup on terrain.

4. CONCLUSION

This paper described a data organization for precision markup modeling in a global geospatial environment. The global environment supports interactive visual navigation from global overviews to details on the ground with rendered precision at the resolution of inches or less. To efficiently handle details over this range of scales while providing accurate placement of objects, a set of nested coordinate systems is used. This coordinate structure supports multi-resolution models of imagery, terrain, vector data, buildings, moving objects, and other geospatial data. Thus objects that are static or moving on the terrain can be displayed without inaccurate positioning or jumping due to coordinate round-off.

REFERENCES

1. Barrus, J.W.; R.C. Waters, D.B. Anderson, Locales: supporting large multiuser virtual environments, Computer Graphics and Applications, IEEE, Volume: 16, Issue: 6, Nov 1996, Page(s): 50-57.

2. Davis, D., Jiang, T.F., Ribarsky, W., Faust, N. Intent, Perception, and Out-of-Core Visualization Applied to Terrain. *Proc. IEEE Visualization '98*, pp. pp. 455-458 (1998).

3. Davis, D., Jiang, T.F., Ribarsky, W., Faust, N., and Ho, S. Real-Time Visualization of Scalably Large Collections of Heterogeneous Objects. *Proc. IEEE Visualization '99*, pp. 437-440 (1999).

4. Justin Jang, William Ribarsky, Chris Shaw, and Nickolas Faust, "View-Dependent Multiresolution Splatting of Non-Uniform Data,", Eurographics-IEEE Visualization Symposium 2002, pp. 125-132.

5. Dave Koller, ,P.Lindstrom, W. Ribarsky, L.F. Hodges, N. Faust, and G. Turner, Virtual GIS: A Real-Time 3D Geographic Information System. *Proc. Visualization '95*, pp. 94-100 (1995).

6. David Krum, Olugbenga Omoteso, Thad Starner, and Larry Hodges, "Speech and Gesture Multimodal Control of a Whole Earth 3D Virtual Environment," Eurographics-IEEE Visualization Symposium 2002 pp. 195-200.

7. David Krum, Rob Melby, William Ribarsky, and Larry Hodges. Isometric Pointer Interfaces for Wearable 3D Visualization. To be published, ACM CHI 2003.

8. Peter Lindstrom, D. Koller, Ribarsky, W., L.F. Hodges, N. Faust, and G. Turner, A. Real-Time, Continuous Level of Detail Rendering of Height Fields. *Proc. SIGGRAPH '96, Computer Graphics*, pp. 109-118 (1996).

9. Peter Lindstrom, W. Ribarsky, N. Faust, and T.Y. Jiang, New Methods for Global Terrain Visualization. GVU Tech. Report 99-35, Georgia Tech (1999).

10. H. Samet, The Quadtree and Related Hierarchical Data Structures. *ACM Comp. Surveys* 16(2), pp. 187-260 (1984).

11. Zachary Wartell, William Ribarsky and Larry F. Hodges, Third-Person Navigation of Whole-Planet Terrain in a Head-tracked Stereoscopic Environment, IEEE} Virtual Reality 1999 Conference, 1999, March 13-17}, IEEE Computer Society Press, pages 141-148.

12. Zachary Wartell, Eunjung Kang, Tony Wasilewski, William Ribarsky, Nickolas Faust, Rendering Vector Data over Global, Multi-resolution 3D Terrain, (*To Appear*) Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2003) G.-P. Bonneau, S. Hahmann, C. D. Hansen (Editors)

13. Tony Wasilewski, Nickolas Faust, William Ribarsky, "From Urban Terrain Models to Visible Cities," IEEE CG&A (2002), Vol. 22(4), pp. 10-15.

14. Java 3D API, http://java.sun.com/products/java-media/3D/index.html.